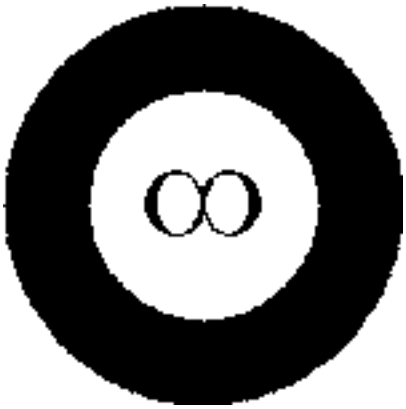


NonlinearRegression

Release 0.1

Feb 19, 2021

Contents:

CHAPTER 1

Introduction

`NonlinearRegression` is a small suit of tools to perform nonlinear regression, *scikitlearn* style. It uses linear regression and data transformation to perform unweighted nonlinear regression and implements a version of function spaces as Hilbert spaces to do weighted nonlinear regression.

Also, has a simple class to cross validate time series when treated as a regression problem.

1.1 Dependencies

- NumPy,
- scipy,
- scikit-learn,

1.2 Download

NonlinearRegression can be obtained from https://github.com/mghasemi/nonlinear-regression.

1.3 Installation

To install *NonlinearRegression*, run the following in terminal:

```
sudo python setup.py install
```

1.4 Documentation

The documentation is produced by Sphinx and is intended to cover code usage as well as a bit of theory to explain each method briefly. For more details refer to the documentation at nonlinearregression.rtfd.io/.

1.5 License

This code is distributed under MIT license:

1.5.1 MIT License

CHAPTER 2

Nonlinear Regression

2.1 From Linear Regression to Nonlinear Regression

Given an arbitrary set of point $\{(X_{i1}, \ldots, X_{in}, Y_i) : i = 1, \ldots, m\}$ and an arbitrary norm $\|\cdot\|$, a *linear regression*, $\ell_{\|\cdot\|}$, with respect to the norm corresponds to a tuple of numbers $(a_1, \ldots, a_n, b)$ such that

$$\sum_{i=1}^{m} \| \sum_{j=1}^{n} a_i X_{ij} + b - Y_i \|$$

is minimized. Generally the norm is taken to be a wighted Euclidean norm. Then the association $\ell_{\|\cdot\|}(x_1, \ldots, x_n) = \sum_{j=1}^{n} a_i x_i + b$ provides an approximation for Y_i's with respect to $\|\cdot\|$.

Suppose that a set of basic functions, $B = \{f_0, \ldots, f_p\}$, a set of data points $X = \{(X_i, Y_i) : i = 1, \ldots, m\}$, and a norm $\|\cdot\|$ are given. A nonlinear regression, $\mathcal{R}_B$, based on B and $\|\cdot\|$ corresponds to a tuple $a_0, \ldots, a_p$ such that

$$\sum_{i=1}^{m} \| \sum_{j=0}^{p} a_j f_j(X_i) - Y_i \|$$

is minimized.

Define a transformer

$$\begin{array}{rccl} T_B : X & \longrightarrow & \mathbb{R}^{p+1} \\ x & \mapsto & (f_0(x), \ldots, f_p(x)). \end{array}$$

Then it is clear that $\mathcal{R}_B(x) = \ell_{\|\cdot\|} \circ T_B(x)$.

2.2 Orthonormal system of functions

Let X be a topological space and μ be a finite Borel measure on X. The bilinear function $\langle \cdot, \cdot \rangle$ defined on $L_2(X, \mu)$ as $\langle f, g \rangle = \int_X fg d\mu$ is an inner product which turns $L_2(X, \mu)$ into a Hilbert space.

Let us denote the family of all continuous real valued functions on a non-empty compact space *X* by $\mathrm{C}(X)$. Suppose that among elements of $\mathrm{C}(X)$, a subfamily *A* of functions are of particular interest. Suppose that *A* is a subalgebra of $\mathrm{C}(X)$ containing constants. We say that an element $f \in \mathrm{C}(X)$ can be approximated by elements of *A*, if for every $\epsilon > 0$, there exists $p \in A$ such that $|f(x) - p(x)| < \epsilon$ for every $x \in X$. The following classical results guarantees when every $f \in \mathrm{C}(X)$ can be approximated by elements of *A*.

Let $(V, \langle \cdot, \cdot \rangle)$ be an inner product space with $\|v\|_2 = \langle v, v \rangle^{\frac{1}{2}}$. A basis $\{v_\alpha\}_{\alpha \in I}$ is called an orthonormal basis for *V* if $\langle v_\alpha, v_\beta \rangle = \delta_{\alpha\beta}$, where $\delta_{\alpha\beta} = 1$ if and only if $\alpha = \beta$ and is equal to *0* otherwise. Every given set of linearly independent vectors can be turned into a set of orthonormal vectors that spans the same sub vector space as the original. The following well-known result gives an algorithm for producing such orthonormal from a set of linearly independent vectors:

Note: Gram–Schmidt

Let $(V, \langle \cdot, \cdot \rangle)$ be an inner product space. Suppose $\{v_i\}_{i=1}^n$ is a set of linearly independent vectors in *V*. Let

$$u_1 := \frac{v_1}{\|v_1\|_2}$$

and (inductively) let

$$w_k := v_k - \sum_{i=1}^{k-1} \langle v_k, u_i \rangle u_i \text{ and } u_k := \frac{w_k}{\|w_k\|_2}.$$

Then $\{u_i\}_{i=1}^n$ is an orthonormal collection, and for each *k*,

$$span\{u_1, u_2, \cdots, u_k\} = span\{v_1, v_2, \cdots, v_k\}.$$

Note that in the above note, we can even assume that $n = \infty$.

Let $B = \{v_1, v_2, \dots\}$ be an ordered basis for $(V, \langle \cdot, \cdot \rangle)$. For any given vector $w \in V$ and any initial segment of *B*, say $B_n = \{v_1, \dots, v_n\}$, there exists a unique $v \in \text{span}(B_n)$ such that $\|w - v\|_2$ is the minimum:

Note: Let $w \in V$ and *B* a finite orthonormal set of vectors (not necessarily a basis). Then for $v = \sum_{u \in B} \langle u, w \rangle u$

$$\|w - v\|_2 = \min_{z \in \text{span}(B)} \|w - z\|_2.$$

Now, let μ be a finite measure on *X* and for $f, g \in \mathrm{C}(X)$ define $\langle f, g \rangle = \int_X fg d\mu$. This defines an inner product on the space of functions. The norm induced by the inner product is denoted by $\|\cdot\|_2$. It is evident that

$$\|f\|_2 \leq \|f\|_\infty \mu(X), \ \forall f \in \mathrm{C}(X),$$

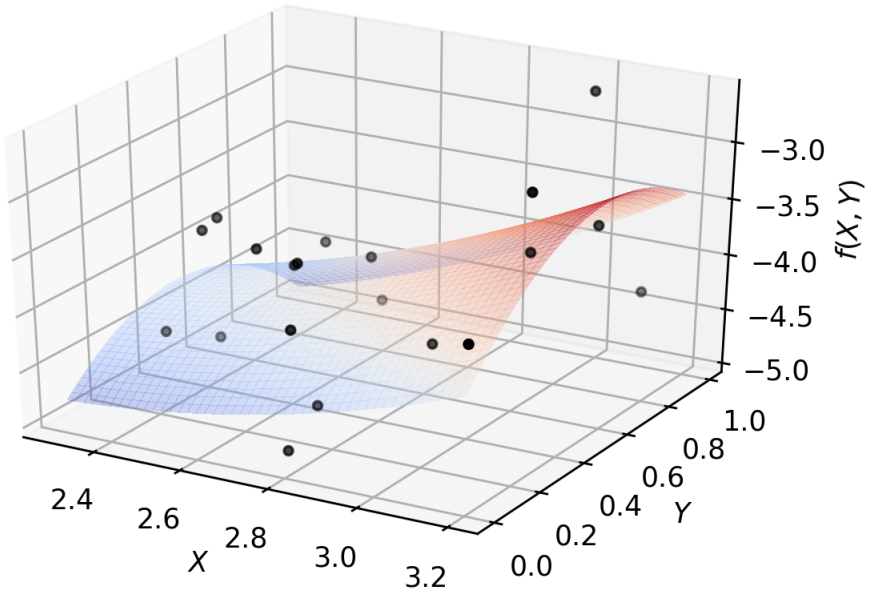
which implies that any good approximation in $\|\cdot\|_\infty$ gives a good $\|\cdot\|_2$-approximation. But generally, our interest is the other way around. Employing Gram-Schmidt procedure, we can find $\|\cdot\|_2$ within any desired accuracy, but this does not guarantee a good $\|\cdot\|_\infty$-approximation. The situation is favorable in finite dimensional case. Take $B = \{p_1, \dots, p_n\} \subset \mathrm{C}(X)$ and $f \in \mathrm{C}(X)$, then there exists $K_f > 0$ such that for every $g \in \text{span}(B \cup \{f\})$,

$$K_f \|g\|_\infty \leq \|g\|_{2 \leq} \|g\|_\infty \mu(X).$$

Since *X* is assumed to be compact, $\mathrm{C}(X)$ is separable, i.e., $\mathrm{C}(X)$ admits a countable dimensional dense subvector space (e.g. polynomials for when *X* is a closed, bounded interval). Thus for every $f \in \mathrm{C}(X)$ and every $\epsilon > 0$ one can find a big enough finite *B*, such that the above inequality holds. In other words, good enough $\|\cdot\|_2$-approximations of f give good $\|\cdot\|_\infty$-approximations, as desired.

Example. Polynomial regression on 2-dimensional random data:

```
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib.pyplot as plt
import numpy as np
from GeneralRegression.NpyProximation import HilbertRegressor, Measure
from GeneralRegression.extras import FunctionBasis
def randrange(n, vmin, vmax):
    '''
    Helper function to make an array of random numbers having shape (n, )
    with each number distributed Uniform(vmin, vmax).
    '''
    return (vmax - vmin)*np.random.rand(n) + vmin

# degree of polynomials
deg = 2
FB = FunctionBasis()
B = FB.poly(2, deg)
# initiate regressor
regressor = HilbertRegressor(base=B)
# number of random points
n = 20
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
for c, m, zlow, zhigh in [('k', 'o', -5, -2.5)]:
    xs = randrange(n, 2.3, 3.2)
    ys = randrange(n, 0, 1.0)
    zs = randrange(n, zlow, zhigh)
    ax.scatter(xs, ys, zs, c=c, s=10, marker=m)
ax.set_xlabel('$X$')
ax.set_ylabel('$Y$')
ax.set_zlabel('$f(X,Y)$')
X = np.array([np.array((xs[_], ys[_])) for _ in range(n)])
y = np.array([np.array((zs[_],)) for _ in range(n)])
X_ = np.arange(2.3, 3.2, 0.02)
Y_ = np.arange(0, 1.0, 0.02)
_X, _Y = np.meshgrid(X_, Y_)
# fit the regressor
regressor.fit(X, y)
# prepare the plot
Z = []
for idx in range(_X.shape[0]):
    _X_ = _X[idx]
    _Y_ = _Y[idx]
    _Z_ = []
    for jdx in range(_X.shape[1]):
        t = np.array([np.array([_X_[jdx], _Y_[jdx]])])
        _Z_.append(regressor.predict(t)[0])
    Z.append(np.array(_Z_))
Z = np.array(Z)
surf = ax.plot_surface(_X, _Y, Z, cmap=cm.coolwarm, linewidth=0, antialiased=False,␣
↪alpha=.3)
```



2.3 Weighted & Unweighted Regression

The following example demonstrates how to use *GenericRegressor* to perform a nonlinear regression based on customized function basis. In the example we use a mixture of polynomials, trigonometric functions and exponential functions of the form $x^k e^{\pm x/\ell}$ to estimate the function $x \times e^{\sin(x^2)} + x^2 \times \cos(x)$.

The confidence interval is the default 95% for points:

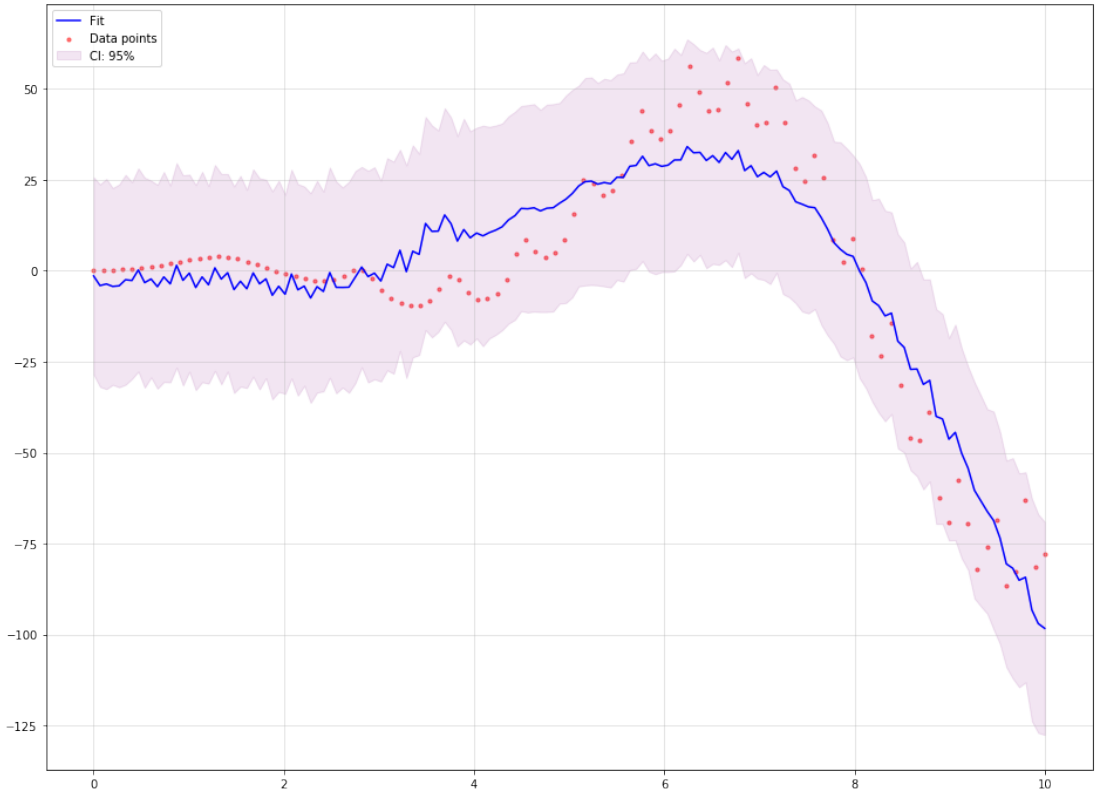
```
from random import randint

import matplotlib.pyplot as plt
import numpy as np
from sklearn.linear_model import BayesianRidge

from GeneralRegression.GeneralRegression import GenericRegressor

plt.figure(randint(1, 1000), figsize=(16, 12))

# Make up a 1-dim regression data
n_samples = 100
f = lambda x: x * np.exp(np.sin(x ** 2)) + np.cos(x) * x ** 2
X = np.linspace(0., 10, n_samples).reshape((-1, 1))
y = f(X).reshape((1, -1))[0]


# Function basis generator
def mixed(X, p_d=3, f_d=1, l=1., e_d=2):
```

(continues on next page)

(continued from previous page)

```
    """
    A mixture of polynomial, Fourier and exponential functions

    :param X: the domain to be transformed
    :param p_d: the maximum degree of polynomials to be included
    :param f_d: the maximum degree of discrete Fourier transform
    :param e_d: the maximum degree of the `x` coefficient to be included as :math:`x^
↪d\times e^{\pm x}`

    :return: the transformed data points
    """
    points = []
    for x in X:
        point = [1.]
        for deg in range(1, f_d + 1):
            point.append(np.sin(deg * x[0] / l))
            point.append(np.cos(deg * x[0] / l))
        for deg in range(1, p_d + 1):
            point.append(x[0] ** deg)
        for deg in range(e_d + 1):
            point.append((x[0] ** deg) * np.exp(-x[0] / l))
            point.append((x[0] ** deg) * np.exp(x[0] / (2.5 * l)))
        points.append(np.array(point))
    return np.array(points)


domain = np.linspace(min(X), max(X), 150)

regressor = GenericRegressor(mixed, regressor=BayesianRidge, **dict(p_d=3, f_d=50,␣
↪l=1., e_d=0))
regressor.fit(X, y)
y_pred = regressor.predict(domain)

plt.scatter(X, y, color='red', s=10, marker='o', alpha=0.5, label="Data points")
plt.plot(domain, y_pred, color='blue', label='Fit')
plt.fill_between(domain.reshape((1, -1))[0],
                 y_pred - regressor.ci_band,
                 y_pred + regressor.ci_band,
                 color='purple',
                 alpha=0.1, label='CI: 95%')
plt.legend(loc=2)
plt.grid(True, alpha=.4)
plt.show()
```

The output looks like the following image:



Now, we use two different weights to approximate the same function. The first weight puts more emphasise on the mid points and less on the extreme points, while the second weight puts less emphasise on the lower values and more on the higher ones. This example shows how to use *HilbertRegressor* and *GeneralRegression.extras.FunctionBasis*.

In contrast to the previous example, the confidence intervals are 95% defaults for the curves, not the points:

```
import numpy as np
import matplotlib.pyplot as plt
from random import randint

from GeneralRegression.NpyProximation import HilbertRegressor, Measure
from GeneralRegression.extras import FunctionBasis

plt.figure(randint(1, 1000), figsize=(16, 12))
# Make up a 1-dim regression data
n_samples = 100
f = lambda x: x * np.exp(np.sin(x**2)) + np.cos(x)*x ** 2
X = np.linspace(0., 10, n_samples).reshape((-1,1))
y = f(X)


def pfe_1d(p_d=3, f_d=1, l=1.):
    basis = FunctionBasis()
    p_basis = basis.poly(1, p_d)
    f_basis = basis.fourier(1, f_d, l)[1:]
    e_basis = []
    return p_basis + f_basis + e_basis

domain = np.linspace(min(X), max(X), 150)
```

(continues on next page)

(continued from previous page)

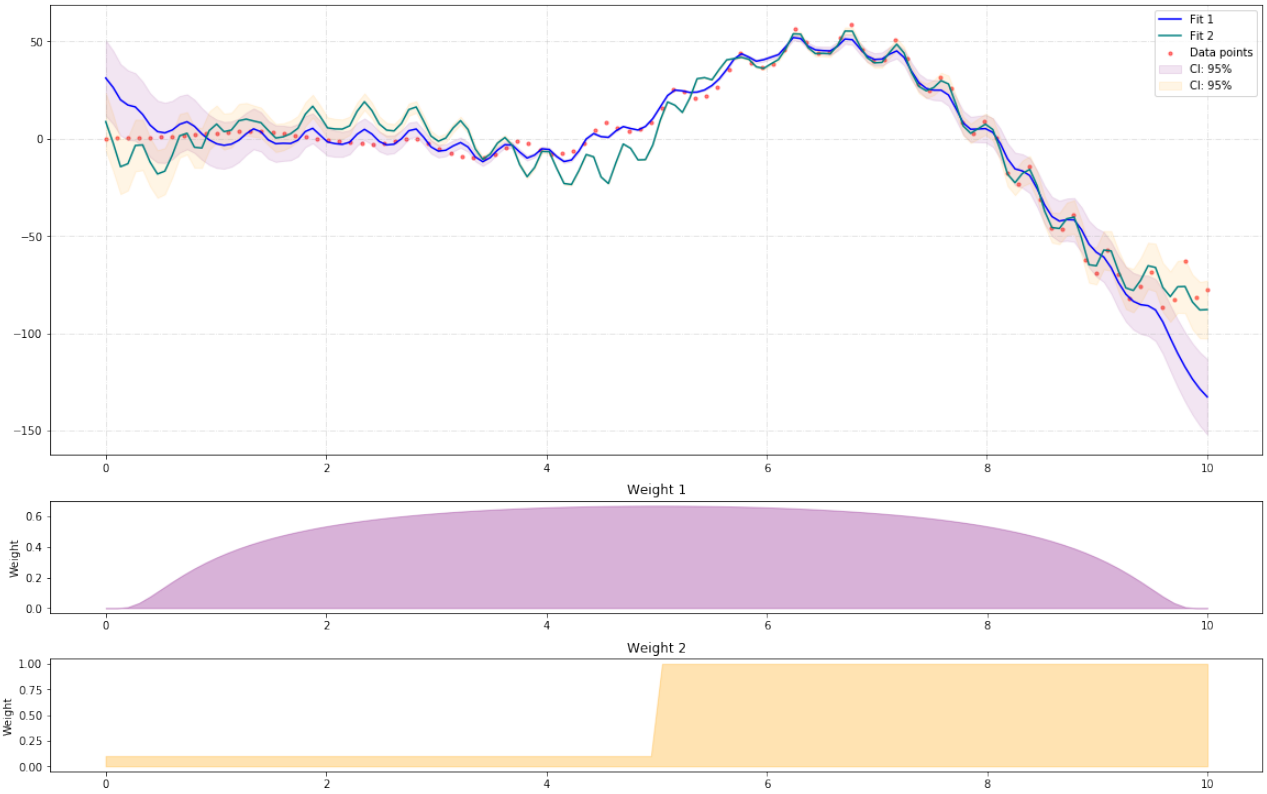
```
x_min, x_max = X.min(), X.max()
x_mid = (x_min + x_max) / 2.
w_min = .1
w_max = 5.
ws1 = {_[0]: np.exp(-1./max(abs((_[0]-x_min)*(_[0]-x_max))/10, 1.e-5))
          for _ in X}
Xs1 = [_[0] for _ in X]
Ws1 = [ws1[_] for _ in Xs1]

ws2 = {_[0]: .1 if _[0] < x_mid else 1.
          for _ in X}
Xs2 = [_[0] for _ in X]
Ws2 = [ws2[_] for _ in Xs2]

meas1 = Measure(ws1)
ell = .7
B1 = pfe_1d(p_d=3, f_d=20, l=ell)

regressor1 = HilbertRegressor(base=B1, meas=meas1)
regressor1.fit(X, y)
y_pred1 = regressor1.predict(domain)

meas2 = Measure(ws2)

regressor2 = HilbertRegressor(base=B1, meas=meas2)
regressor2.fit(X, y)
y_pred2 = regressor2.predict(domain)

fig = plt.figure(randint(1, 10000), constrained_layout=True, figsize=(16, 10))
gs = fig.add_gridspec(6, 1)
f_ax1 = fig.add_subplot(gs[:4, :])
f_ax1.scatter(X, y, color='red', s=10, marker='o', alpha=0.5, label="Data points")
f_ax1.plot(domain, y_pred1, color='blue', label='Fit 1')
f_ax1.plot(domain, y_pred2, color='teal', label='Fit 2')
f_ax1.fill_between(domain.reshape((1, -1))[0],
                   y_pred1 - regressor1.ci_band,
                   y_pred1 + regressor1.ci_band,
                   color='purple',
                   alpha=0.1, label='CI: 95%')
f_ax1.fill_between(domain.reshape((1, -1))[0],
                   y_pred2 - regressor2.ci_band,
                   y_pred2 + regressor2.ci_band,
                   color='orange',
                   alpha=0.1, label='CI: 95%')
f_ax1.legend(loc=1)
f_ax1.grid(True, linestyle='-.', alpha=.4)

f_ax2 = fig.add_subplot(gs[4, :])
f_ax2.set_title('Weight 1')
f_ax2.fill_between(Xs1, [0. for _ in Ws1], Ws1, label='Distibution', color='purple',␣
↪alpha=.3)
f_ax2.set_ylabel('Weight')

f_ax3 = fig.add_subplot(gs[5:, :])
f_ax3.set_title('Weight 2')
f_ax3.fill_between(Xs2, [0. for _ in Ws2], Ws2, label='Distibution', color='orange',␣
↪alpha=.3)
```

(continues on next page)

(continued from previous page)

```
f_ax3.set_ylabel('Weight')
```

The output looks like the following image:



Note: The major code difference between *GenericRegressor* and *HilbertRegressor* lies in the way they accept the basis functions. The *funcs* parameter of *GenericRegressor* applies the set of basis function on the input points and returns a new set of data points. While *HilbertRegressor* uses a set of functions as *base* to perform the required calculations.

2.4 Time series as regression with missing steps

The following example illustrates how a typical time series problem with missing data can be treated as a regression problem while using all existing data points.

Using unweighted nonlinear regression:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from random import randint
from sklearn.linear_model import BayesianRidge

from GeneralRegression.GeneralRegression import GenericRegressor
from GeneralRegression.extras import Time2Interval
```

(continues on next page)

(continued from previous page)

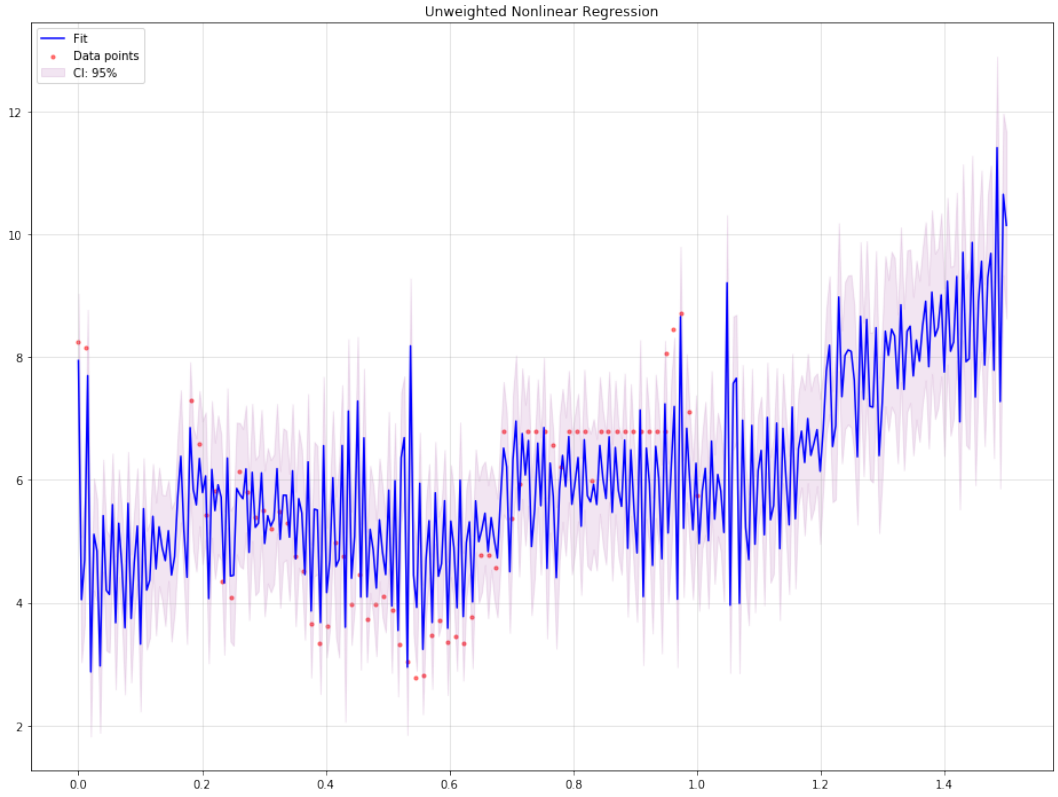
```
df = pd.read_csv("./data/elec_cost.csv", parse_dates=['Effective Start Date (mm/dd/
↪yyyy)'], infer_datetime_format=True)
dm, dx = df['Effective Start Date (mm/dd/yyyy)'].min(), df['Effective Start Date (mm/
↪dd/yyyy)'].max()
time_trans = Time2Interval(dm, dx)
df['T'] = df.apply(lambda x: time_trans.date2num(x['Effective Start Date (mm/dd/yyyy)
↪']), axis=1)

plt.figure(randint(1, 1000), figsize=(16, 12))

X = df[['T']].values
y = df['Cost'].values

def mixed(X, p_d=3, f_d=1, l=1., e_d=2):
    """
    A mixture of polynomial, Fourier and exponential functions

    :param X: the domain to be transformed
    :param p_d: the maximum degree of polynomials to be included
    :param f_d: the maximum degree of discrete Fourier transform
    :param e_d: the maximum degree of the `x` coefficient to be included as :math:`x^
↪d\times e^{\pm x}`

    :return: the transformed data points
    """
    points = []
    for x in X:
        point = []
        point.append(1.)
        for deg in range(1, f_d + 1):
            point.append(np.sin(deg * x[0] / l))
            point.append(np.cos(deg * x[0] / l))
        for deg in range(1, p_d + 1):
            point.append(x[0] ** deg)
        for deg in range(e_d + 1):
            point.append((x[0] ** deg) * np.exp(-x[0] / l))
            point.append((x[0] ** deg) * np.exp(x[0] / (2.5 * l)))
        points.append(np.array(point))
    return np.array(points)

domain = np.linspace(min(X), max(X)+.5, 300)

regressor = GenericRegressor(mixed, regressor=BayesianRidge, **dict(p_d=5, f_d=50,␣
↪l=1./12., e_d=-1))
regressor.fit(X, y)
y_pred = regressor.predict(domain)

plt.scatter(X, y, color='red', s=10, marker='o', alpha=0.5, label="Data points")
plt.plot(domain, y_pred, color='blue', label='Fit')
plt.fill_between(domain.reshape((1, -1))[0],
                 y_pred - regressor.ci_band,
                 y_pred + regressor.ci_band,
                 color='purple',
                 alpha=0.1, label='CI: 95%')
plt.legend(loc=2)
plt.grid(True, alpha=.4)
```

(continues on next page)

(continued from previous page)

```
plt.title('Unweighted Nonlinear Regression')
```

The output would be tha following:



Using weighted nonlinear regression:

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from random import randint
from sklearn.linear_model import BayesianRidge

from GeneralRegression.NpyProximation import HilbertRegressor, Measure
from GeneralRegression.extras import FunctionBasis

df = pd.read_csv("./data/elec_cost.csv", parse_dates=['Effective Start Date (mm/dd/
↪yyyy)'], infer_datetime_format=True)
dm, dx = df['Effective Start Date (mm/dd/yyyy)'].min(), df['Effective Start Date (mm/
↪dd/yyyy)'].max()
time_trans = Time2Interval(dm, dx)
df['T'] = df.apply(lambda x: time_trans.date2num(x['Effective Start Date (mm/dd/yyyy)
↪']), axis=1)

plt.figure(randint(1, 1000), figsize=(16, 12))

X = df[['T']].values
y = df['Cost'].values

def pfe_1d(p_d=3, f_d=3, l=1.):
```

(continues on next page)

(continued from previous page)

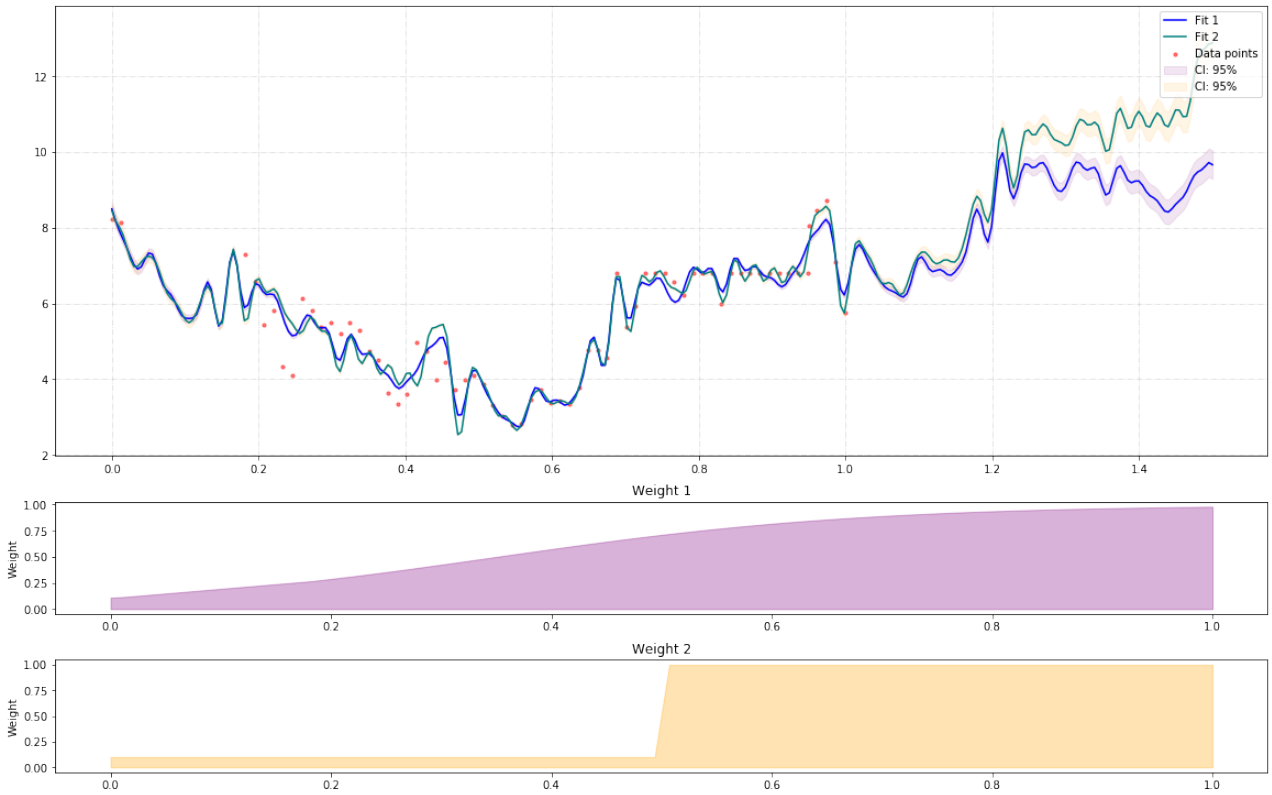
```
    basis = FunctionBasis()
    p_basis = basis.poly(1, p_d)
    f_basis = basis.fourier(1, f_d, l)[1:]
    e_basis = []
    return p_basis + f_basis + e_basis

domain = np.linspace(min(X), max(X)+.5, 300)

x_min, x_max = X.min(), X.max()
x_mid = (x_min + x_max) / 2.
w_min = .1
w_max = 5.
ws1 = {_[0]: 1./(1. + np.exp(-6*(_[0] - .35)))
          for _ in X}
Xs1 = [_[0] for _ in X]
Ws1 = [ws1[_] for _ in Xs1]
ws2 = {_[0]: .1 if _[0] < x_mid else 1.
          for _ in X}
Xs2 = [_[0] for _ in X]
Ws2 = [ws2[_] for _ in Xs2]
meas1 = Measure(ws1)
ell = 1./12
B1 = pfe_1d(p_d=4, f_d=20, l=ell)

regressor1 = HilbertRegressor(base=B1, meas=meas1)
regressor1.fit(X, y)
y_pred1 = regressor1.predict(domain)

meas2 = Measure(ws2)

regressor2 = HilbertRegressor(base=B1, meas=meas2)
regressor2.fit(X, y)
y_pred2 = regressor2.predict(domain)

fig = plt.figure(randint(1, 10000), constrained_layout=True, figsize=(16, 10))
gs = fig.add_gridspec(6, 1)
f_ax1 = fig.add_subplot(gs[:4, :])
f_ax1.scatter(X, y, color='red', s=10, marker='o', alpha=0.5, label="Data points")
f_ax1.plot(domain, y_pred1, color='blue', label='Fit 1')
f_ax1.plot(domain, y_pred2, color='teal', label='Fit 2')
f_ax1.fill_between(domain.reshape((1, -1))[0],
                   y_pred1 - regressor1.ci_band,
                   y_pred1 + regressor1.ci_band,
                   color='purple',
                   alpha=0.1, label='CI: 95%')
f_ax1.fill_between(domain.reshape((1, -1))[0],
                   y_pred2 - regressor2.ci_band,
                   y_pred2 + regressor2.ci_band,
                   color='orange',
                   alpha=0.1, label='CI: 95%')
f_ax1.legend(loc=1)
f_ax1.grid(True, linestyle='-.', alpha=.4)

f_ax2 = fig.add_subplot(gs[4, :])
f_ax2.set_title('Weight 1')
f_ax2.fill_between(Xs1, [0. for _ in Ws1], Ws1, label='Distibution', color='purple',␣
↪alpha=.3)
```

(continues on next page)

(continued from previous page)

```
f_ax2.set_ylabel('Weight')

f_ax3 = fig.add_subplot(gs[5:, :])
f_ax3.set_title('Weight 2')
f_ax3.fill_between(Xs2, [0. for _ in Ws2], Ws2, label='Distibution', color='orange',␣
↪alpha=.3)
f_ax3.set_ylabel('Weight')
```

The output would be tha following:



And for what it worth

Using support vector regression with rbf:

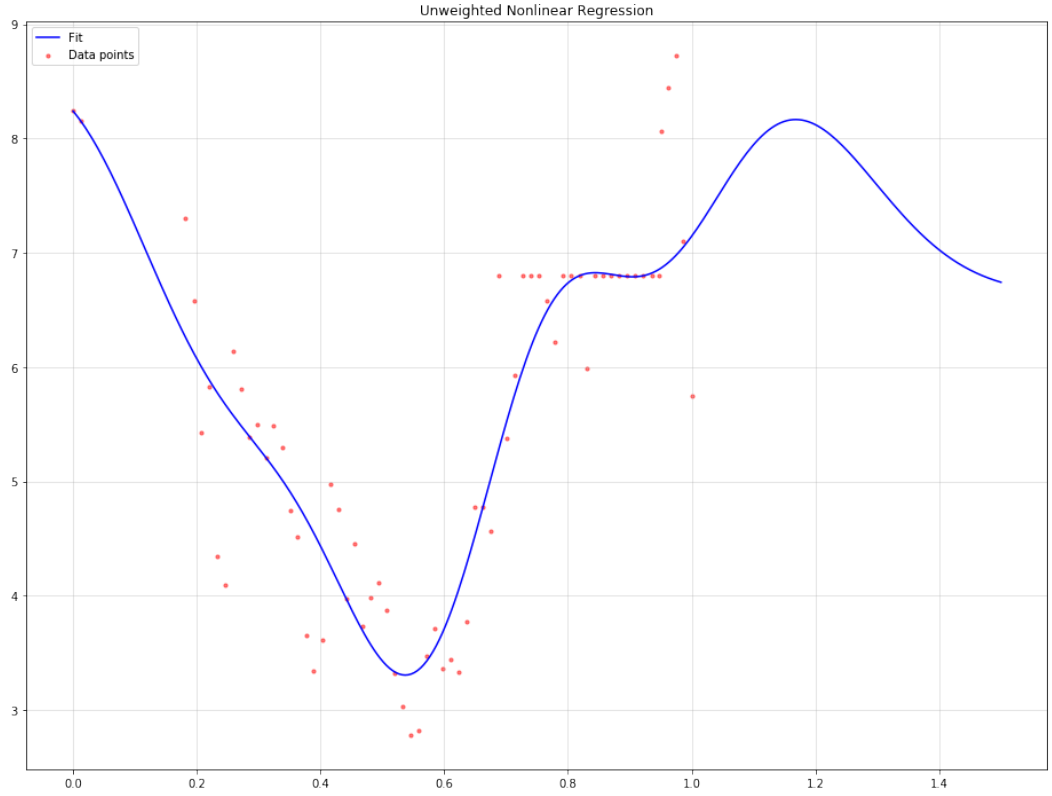
```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from random import randint
from sklearn.svm import SVR

from GeneralRegression.extras import Time2Interval

df = pd.read_csv("./data/elec_cost.csv", parse_dates=['Effective Start Date (mm/dd/
↪yyyy)'], infer_datetime_format=True)
dm, dx = df['Effective Start Date (mm/dd/yyyy)'].min(), df['Effective Start Date (mm/
↪dd/yyyy)'].max()
time_trans = Time2Interval(dm, dx)
df['T'] = df.apply(lambda x: time_trans.date2num(x['Effective Start Date (mm/dd/yyyy)
↪']), axis=1)
```

(continues on next page)

(continued from previous page)

```
plt.figure(randint(1, 1000), figsize=(16, 12))

X = df[['T']].values
y = df['Cost'].values

domain = np.linspace(min(X), max(X)+.5, 300)

svr = SVR(kernel='rbf', tol=.0000001, C=10., epsilon=.0001)
svr.fit(X, y)
y_pred = svr.predict(domain)

plt.scatter(X, y, color='red', s=10, marker='o', alpha=0.5, label="Data points")
plt.plot(domain, y_pred, color='blue', label='Fit')
plt.legend(loc=2)
plt.grid(True, alpha=.4)
plt.title('Unweighted Nonlinear Regression')
```

The output would be tha following:



CHAPTER 3

Evaluating forecast models

- It is common practice to separate the data into two portions, **training** and **test** data.
- The training data is used to estimate any parameters of a forecasting method and the test data is used to evaluate its accuracy
- Since the test data is not used in determining the forecasts, it should provide a reliable indication of how well the model is likely to forecast on new data.

The size of the test set is typically about 20% of the total sample, although this value depends on how long the sample is and how far ahead you want to forecast.

The test set should ideally be at least as large as the maximum forecast horizon required.

The *ModelSelection.TimeSeriesCV* provides a *scikit-learn* compatible interface for cross-validation of time series.

CHAPTER 4

Code Documentation

class GeneralRegression.**GenericRegressor**(*funcs*, *regressor=None*, *ci=0.95*, ***kwargs*)

Uses a linear regression algorithm and a transformer to perform nonlinear regression. Using a set of functions $(f_0, \ldots, f_n)$, and a point x, lifts the point x to $(x, f_0(x), \ldots, f_n(x))$ and applies a linear regression. The result will be a nonlinear regression based on $f_0, \ldots, f_n$.

Parameters

- **funcs** – a function that transforms the data points
- **regressor** – the linear regression method which should be scikit-learn compatible; default: *BayesianRidge*
- **ci** – confidence interval; float between 0 and 1.
- **kwargs** – argument to be passed to *funcs*

fit(*X*, *y*)

Calculates an orthonormal basis according to the given function basis and the linear regressor..

Parameters

- **X** – Training data
- **y** – Target values

Returns *self*

predict(*X*)

Predict using the Hilbert regression method

Parameters **X** – data points for prediction

Returns returns predicted values

4.1 Hilbert Space based regression

exception NpyProximation.**Error**(**args*)
Generic errors that may occur in the course of a run.

class NpyProximation.**FunctionSpace**(*dim=1*, *measure=None*, *basis=None*)
A class tha facilitates a few types of computations over function spaces of type $L_2(X, \mu)$

Parameters

- **dim** – the dimension of 'X' (default: 1)
- **measure** – an object of type *Measure* representing μ
- **basis** – a finite basis of functions to construct a subset of $L_2(X, \mu)$

form_basis()
Call this method to generate the orthogonal basis corresponding to the given basis. The result will be stored in a property called `orth_base` which is a list of function that are orthogonal to each other with respect to the measure `measure` over the given range `domain`.

inner(*f*, *g*)
Computes the inner product of the two parameters with respect to the measure *measure*, i.e., $\int_X f \cdot g d\mu$.

Parameters

- **f** – callable
- **g** – callable

Returns the quantity of $\int_X f \cdot g d\mu$

project(*f*, *g*)
Finds the projection of f on g with respect to the inner product induced by the measure *measure*.

Parameters

- **f** – callable
- **g** – callable

Returns the quantity of $\frac{\langle f,g\rangle}{\|g\|_2} g$

series(*f*)
Given a function *f*, this method finds and returns the coefficients of the series that approximates f as a linear combination of the elements of the orthogonal basis B. In symbols $\sum_{b\in B}\langle f, b\rangle b$.

Returns the list of coefficients $\langle f, b\rangle$ for $b \in B$

class NpyProximation.**HilbertRegressor**(*deg=3*, *base=None*, *meas=None*, *f_space=None*, *c_limit=0.95*)
Regression using Hilbert Space techniques Scikit-Learn style.

Parameters

- **deg** – int, default=3 The degree of polynomial regression. Only used if *base* is *None*
- **base** – list, default = None a list of function to form an orthogonal function basis
- **meas** – *NpyProximation.Measure*, default = None the measure to form the $L_2(\mu)$ space. If *None* a discrete measure will be constructed based on *fit* inputs
- **f_space** – *NpyProximation.FunctionBasis*, default = None the function subspace of $L_2(\mu)$, if *None* it will be initiated according to *self.meas*
- **c_limit** – for confidence interval

- **apprx** – It is a callable, this will be constructed on *fit* method. It use for approximate after fitting/learning.

fit (*X*, *y*)
Calculates an orthonormal basis according to the given function space basis and the discrete measure from the training points.

Parameters

- **X** – Training data
- **y** – Target values

Returns *self*

predict (*X*)
Predict using the Hilbert regression method

Parameters **X** – data points for prediction

Returns returns predicted values

score (*X*, *y*, *sample_weight=None*)
The default scoring method is the weighted mean square error

Parameters

- **X** –
- **y** –
- **sample_weight** –

Returns

class NpyProximation.**Measure** (*density=None*, *domain=None*)
Constructs a measure μ based on *density* and *domain*.

Parameters

- **density** – the density over the domain: + if none is given, it assumes uniform distribution
  - if a callable *h* is given, then $d\mu = h(x)dx$
  - if a dictionary is given, then $\mu = \sum w_x \delta_x$ a discrete measure. The points x are the keys of the dictionary (tuples) and the weights w_x are the values.
- **domain** – if *density* is a dictionary, it will be set by its keys. If callable, then *domain* must be a list of tuples defining the domain's box. If None is given, it will be set to $[-1, 1]^n$

integral (*f*)
Calculates $\int_{domain} f d\mu$.

Parameters **f** – the integrand

Returns the value of the integral

norm (*p*, *f*)
Computes the norm-*p* of the *f* with respect to the current measure, i.e., $(\int_{domain} |f|^p d\mu)^{1/p}$.

Parameters

- **p** – a positive real number
- **f** – the function whose norm is desired.

Returns $\|f\|_{p,\mu}$

class NpyProximation.**Regression**(*points*, *dim=None*)

Given a set of points, i.e., a list of tuples of the equal lengths *P*, this class computes the best approximation of a function that fits the data, in the following sense:

- if no extra parameters is provided, meaning that an object is initiated like `R = Regression(P)` then calling `R.fit()` returns the linear regression that fits the data.
- if at initiation the parameter *deg=n* is set, then `R.fit()` returns the polynomial regression of degree *n*.
- if a basis of functions provided by means of an *OrthSystem* object (`R.SetOrthSys(orth)`) then calling `R.fit()` returns the best approximation that can be found using the basic functions of the *orth* object.

Parameters

- **points** – a list of points to be fitted or a callable to be approximated
- **dim** – dimension of the domain

fit()

Fits the best curve based on the optional provided orthogonal basis. If no basis is provided, it fits a polynomial of a given degree (at initiation) :return: The fit.

set_func_spc(*sys*)

Sets the bases of the orthogonal basis

Parameters **sys** – *orthsys.OrthSystem* object.

Returns None

Note: For technical reasons, the measure needs to be given via *set_measure* method. Otherwise, the Lebesque measure on $[-1, 1]^n$ is assumed.

---

set_measure(*meas*)

Sets the default measure for approximation.

Parameters **meas** – a measure.Measure object

Returns None

4.2 Time Series Tools

class ModelSelection.**TimeSeriesCV**(*test_ratio=0.2*, *train_ratio=None*, *index=0*)

This is a very naive cross validator for time series. It simply sorts the given index (default 0) and splits the sorted index into a train and a test index set according to the given ratios.

Parameters

- **test_ratio** – (default .2) float betweem 0. and 1., the portion of test data
- **train_ratio** – (default *None*-> .8) float betweem 0. and 1., the portion of train data
- **index** – (default 0) the index of the column that corresponds to a time parameter in the data

get_n_splits(*X=None*, *y=None*, *groups=None*)

Returns the number of splitting iterations in the cross-validator

Parameters

- **X** – Always ignored, exists for compatibility.

- **y** – Always ignored, exists for compatibility.
- **groups** – Always ignored, exists for compatibility.

Returns Returns the number of splitting iterations in the cross-validator which is 1 for time series.

split (*X*, *y=None*, *groups=None*)

Generate indices to split data into training and test set.

Parameters

- **X** – array-like of shape (n_samples, n_features) Training data, where n_samples is the number of samples and n_features is the number of features.
- **y** – array-like of shape (n_samples,), default=None The target variable for supervised learning problems.
- **groups** – array-like of shape (n_samples,), default=None Group labels for the samples used while splitting the dataset into train/test set.

Returns *train* The training set indices for that split. *test* The testing set indices for that split.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index